

Sistemas de Memoria

Índice

1. *Introducción*
2. *Memoria Caché*
3. *Memoria Principal*
4. *Memoria Virtual*

1. Introducción

El sistema de memoria (SM) es un componente crítico del sistema, ya que residen en el mismo los programas y datos que vamos a ejecutar.

El tiempo de ejecución de un programa está muy influenciada por este componente, por la velocidad a la que se accede a datos e instrucciones (al menos un fetch por instrucción y varios accesos a datos posibles).

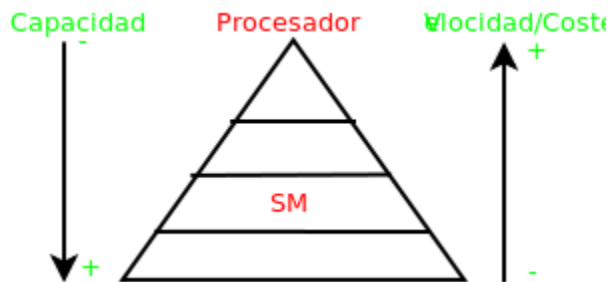
Ej: 1 GHz \rightarrow 1 ns/instr , Mp 50 ns. Con esto “solamente” tenemos 50 ns de Fetch y las instrucciones de Ejecución. No funciona.

Se necesita:

- Más capacidad: por las aplicaciones y el S.O, cada vez más grandes. Además hay que recordar que la memoria se comparte entre todos.
- Más velocidad: los procesadores son cada vez más rápidos (\approx 3GHz) y más numerosos (multicore). Tienen una elevada frecuencia de reloj (pipeline, escalar) y por tanto un tiempo de ciclo reducido. Pero las memorias no han ido a la par (RAM Dinámicas) por lo que tenemos un desequilibrio entre los tiempos de acceso y ejecución.
- Menor coste: buena relación entre coste/prestaciones, no pueden usarse memorias “rápidas” como Mp.

Jerarquía de Memoria

La solución es emplear varios dispositivos con distintas características.

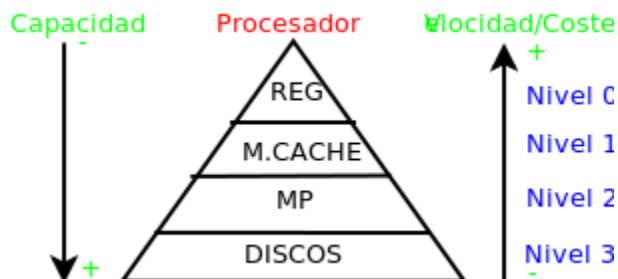


	<i>T_{acceso}</i>	<i>Capacidad</i>
<i>REGISTROS</i>	0.25 – 0.5 ns	< 1 KB
<i>M.CACHE</i>	0.5 – 10 ns	128 KB – 16 MB
<i>M.P.</i>	40 – 100 ns	< 512 GB
<i>DISCOS</i>	5 – 20 ms	> 1 TB

Otros dispositivos

- *Cintas magnéticas: DVDs, CDs, ZIPs (Backup).* Manual, no es lo mismo.
- *Caches de disco:* se usa parte de la Mp para reducir el nº de accesos a disco no es un nuevo dispositivo (Mp).
- *Caches multinivel*

Entonces el esquema queda:



Los registros los gestiona el compilador (SW), nos nos interesa.

Entre el nivel 2 y 3 la gestión la realiza el S.O (SW) pero con soporte HW (Ej: TGB)

El objetivo es obtener la velocidad de la M.Caché con la capacidad del Disco.

*Conceptos básicos de funcionamiento***a) ¿Qué información hay en cada nivel?**

Cada nivel I tiene un subconjunto de la información del nivel $I + 1$ (Propiedad de inclusión)

$$Inf_i < Inf_{i+1} < Inf_{i+2}$$

Hay distintas copias de la misma información en distintos niveles. Esta genera problemas de coherencia entre copias.

Es decir, el mismo dato puede tener valores distintos en cada nivel.

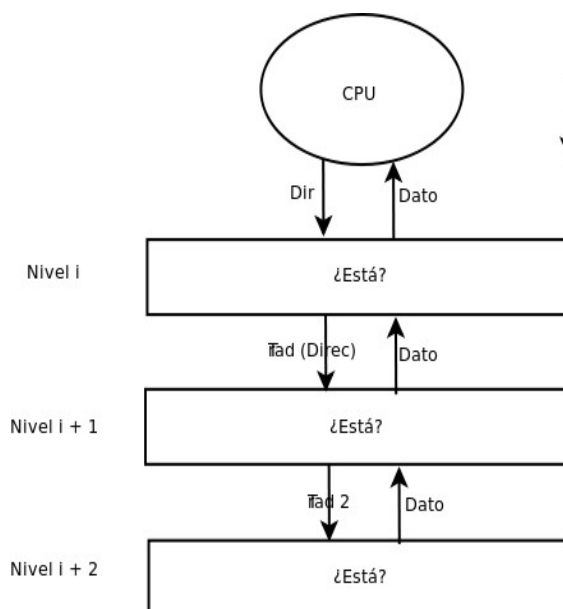
b) La misma información (dato) tiene direcciones distintas en cada nivel por lo que necesitamos traducir la dirección para cambiar de nivel.

Ej: Mp 0 .. 2^n n 40 bits ¿ 2^{30} en M.Caché?
 Mcache 0 .. 2^m m 20 bits

Debido al tamaño tan distinto de cada dispositivo las ubicaciones cambiarán forzosamente y con ello las direcciones.

c) Funcionamiento del S.M.

La información va a moverse de forma matemática entre los distintos niveles subiendo y bajando por la jerarquía. La información estará más cerca de la CPU cuanto más se utilice.



La CPU pregunta por datos e instrucciones (mediante sus direcciones). Si están en el nivel superior (acierto) se devuelve dicho dato, si no está (Fallo) se traduce la dirección y se baja al siguiente nivel donde se repite la pregunta.

Cuando se encuentra un nivel n se sube el dato por toda la jerarquía hasta CPU está implica actualizar los niveles superiores.

Para dicha actualización necesitamos:

- **Política de ubicación:** donde alojarlo.
- **Política de reemplazo:** si está todo ocupado, ¿cuál desalojamos?
- **Política de escritura:** si el dato está modificado, ¿cómo actualizamos niveles inferiores?

Obs: La información siempre viaja en bloques, nunca de forma individual para reducir los costes. El tamaño del bloque es proporcional al coste.

Obs: La información siempre viaja entre niveles consecutivos, nunca se salta un nivel.

La información va ascendiendo por la jerarquía a medida que se utilice y cuando deje de utilizarse será desalojada por otra información mas necesaria (mientras se utilice permanecerá en el nivel más alto).

Ej: Suma de N elementos de un vector

```
Suma = 0;
I = 0;

while (I ≤ N){
    Suma = Suma + vector[I];
    I = I + 1;
}
```

En la primera referencia nos llevamos la información (Instrucción + Dato) al nivel más alto y el resto de accesos se hacen a la máxima velocidad.

Este esquema funciona porque las direcciones o referencias tienden a repetirse.

Proximidad de referencia, propiedad de todos los programas.

Hay dos tipos:

- **Proximidad de referencia temporal:** se tiende a hacer referencia a las mismas direcciones que se emplearán en un pasado reciente. Se debe a bucles y a variables locales.
- **Proximidad de referencia espacial:** se tiende a hacer referencia a direcciones próximas a las que se emplean en un pasado reciente (variables multidimensionales y la ejecución secuencial).

d) Eficiencia del S.M.

Se mide con la *tasa de aciertos* (Hit ratio)

$$Hr = \frac{\text{aciertos nivel } i}{\text{accesos nivel } i} \cdot 100$$

Depende de:

- Tamaño de bloque
- Tamaño del nivel I (Capacidad)
- Política de reemplazo
- Política de escritura
- Programa que se ejecuta

Tiempo de acceso efectivo o tiempo medio de acceso (eficiencia S.M)

Depende del HW de cada nivel y del tiempo de acceso de cada nivel.

Ej: Consideramos una Mcache y la Mp

$$t_{efect} = Hr_{Mca} \cdot t_{acierto} + (1 - Hr_{Mca}) \cdot t_{fallo}$$

Obs: este tiempo es distinto al tiempo de ocupación. El tiempo de acceso efectivo es el tiempo medio desde que la CPU pida un dato hasta que lo obtiene y puede continuar.

El tiempo de ocupación es el tiempo que pasa hasta que el S.M que puede atender una nueva petición, por ejemplo, debido al trasiego de información entre niveles.

2. Memoria Caché

Dispositivo de alta velocidad que se intercala entre CPU y Mp.

Contiene parte de la información presente en memoria principal (actualmente en uso) y su objetivo es que el tiempo medio de acceso sea próximo al tiempo de acceso de la memoria caché (accesos rápidos)

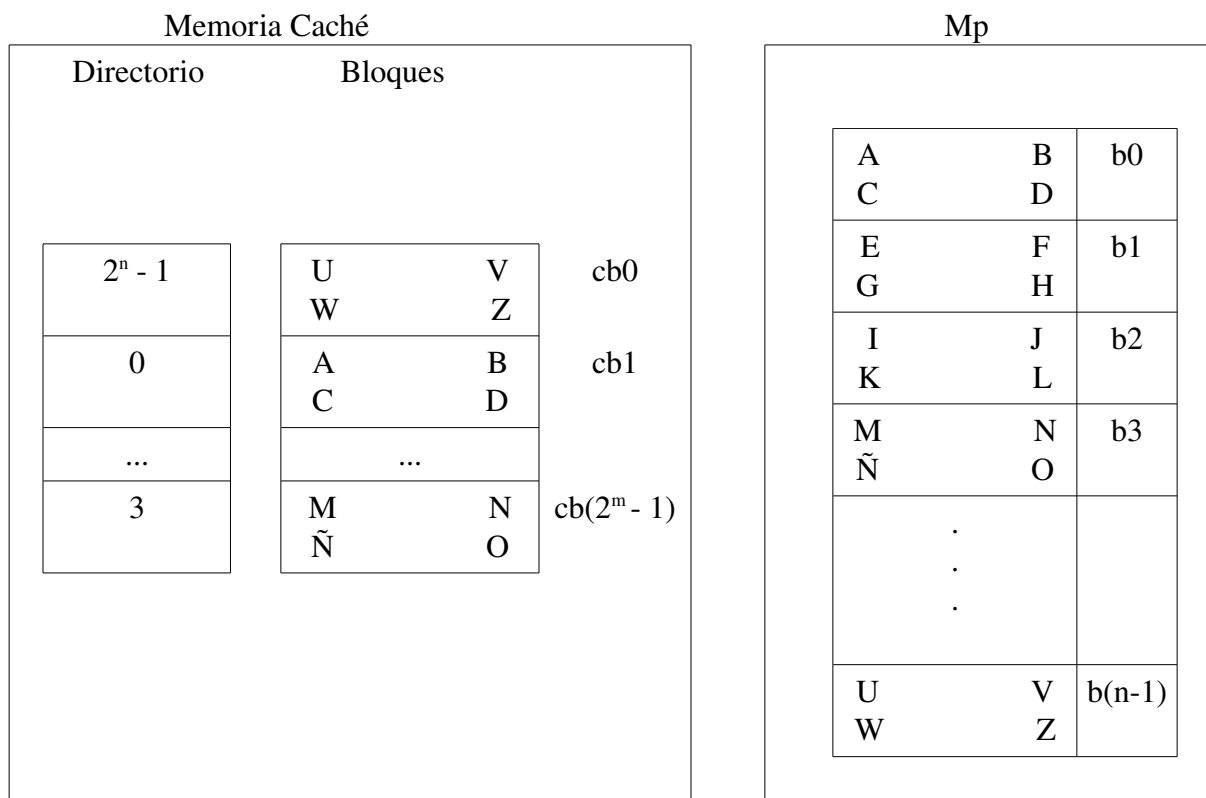
$$t_{efect} = Hr_{Mca} \cdot t_{acierto} + (1 - Hr_{Mca}) \cdot t_{fallo}$$

Debemos minimizar $t_{acierto}$, t_{fallo} y $(1 - Hr)$.

Desde el punto de vista de la memoria caché la Mp está organizada en bloques de información compuestos por conjuntos de direcciones contiguos de **tamaño fijo**.

La Memoria Caché consta de:

- **Bloques (líneas):** tenemos copias de la información presente en memoria principal siempre del mismo tamaño 16 – 64 Bytes.
- **Directorios:** establece la correspondencia entre la Memoria Caché y la Memoria Principal. Se conocen también como etiquetas. Identifica los bloques de Mp presentes en M.Caché.



$$m \ll n$$

Desde el punto de vista de la M_{ca} la diremitida por la CPU para acceder al S.M. Se puede dividir en campos.

Nº de bloque de Mp	Byte / Palabra en el bloque
--------------------	-----------------------------

Obs: la traducción de la dirección en la jerarquía de niveles de memoria no se hace al pasar de M_{ca} a Mp sino directamente para acceder a la M_{ca} por razones históricas.

Entonces, dada una dirección generada por la CPU obtendremos el bloque al que hace referencia (dividiendo la Dirección en campos) y luego buscamos dicho bloque en la Mca.

Ej: Suponiendo direccionamiento a nivel de palabra, la CPU pide leer el dato en la Dir13

0	A	Hay que devolverle la “N”,
1	B	$2^2 = 4$ Datos
2	C	
3	D	
4	E	La posición 13 en binario es 0...0011 01 en el que los dos últimos bits son los que representan la palabra en el bloque. El resto representa el bloque.
5	F	Se busca por tanto el bloque 3 en Memoria Caché → Acierto, devolvemos el dato “01” (va de 0 a 3) que es “N”.
6	G	
7	H	
8	I	
9	J	
10	K	
11	L	
12	M	
13	N	

Si el bloque que buscamos está en Mca es un acierto (existe una etiqueta en el directorio con ese nº de bloques), se accede a la información.

Si no se encuentra, es un fallo. Accedemos a Mp, actualizamos Mca y quizás reemplacemos un dato.

Hay tres tipos de fallos principales:

- **Fallos forzosos:** información que nunca ha estado en Mca. Inevitables.
- **Conflictos:** varios bloques de Mp se ubican en el mismo bloque de Mca, desalojándose mutuamente.
- **Capacidad:** todos los bloques actualmente en uso no caben en Mca y alguno debe dejar la Mca.

Inicialmente el directorio puede contener cualquier información (basura) para evitar aciertos casuales se añade un bit de validez inicialmente a falso y se pone a verdadero cuando se lleve información a dicho bloque.

Hay tres posibles formas de establecer la correspondencia entre los bloques de Mp y Mca, según la política de ubicación empleada.

- **Directa:** un bloque de Mp puede ubicarse en un bloque de Mca
- **Asociativa:** un bloque de Mp puede ubicarse en cualquier bloque de Mca
- **Asociativa por conjuntos:** por conjuntos la Mca se divide a conjuntos de bloques, un bloque de Mp se puede ubicar en cualquier bloque de un único conjunto.

Ej: Para analizar estas tres políticas usamos

- Mca: 8 Kbytes, Bloque 16 bytes $\rightarrow \frac{8 \text{ Kbytes}}{16 \text{ bytes/bloq}} = 2^9 \text{ bloq} = 512$
- Mp: dir 32 bits $\rightarrow \frac{2^{32} \text{ bytes}}{2^9 \text{ bytes/bloq}} = 2^{23} \text{ bloq} \approx 256 \text{ millones}$

Compararemos t_{acierto} , Hr, coste y flexibilidad.

Directa

A cada bloque I de Mp le hacemos corresponder un único bloque J de Mca donde $J = I \bmod C$ siendo C el nº de bloques de Mca.

En el ejemplo: $J = I \bmod 512$

La búsqueda de un bloque en Mca es muy sencillo, vamos al único sitio donde puede estar y si está acierto sino fallo.

Como hay 2^{28} bloques de Mp y 2^9 bloques de Mca, a cada bloque de Mca le corresponden $2^{28}/2^9 = 2^{19}$ bloques de Mp.

Si varios de esos bloques están en uso, se van a desplazar mutuamente: conflictos

Campos de la direc.

19 bits	9 bits	4 bits
Etiqueta	Bloque	Byte bloque

Al acceder al bloque de Mca se compara nuestra dirección con la etiqueta presente para determinar si hay acierto o fallo.

Ventajas

- Se puede acceder simultáneamente a la información y a la etiqueta, minimizando el tiempo en caso de acierto. Si hay fallo se desecha el dato leído
- Política reemplazo inmediata y sencilla.
- Coste bajo: un comparador y todo sencillo
- Fácil de implementar

Desventajas

- Política reemplazonada flexible, se pueden desalojar bloques en uso → conflicto. Ej: si dos vectores coinciden en el mismo bloque todo son fallos.

```
for (i=0...N)           dir a 0x100...0
    C[i]=a[i]*b[i]      dir b 0x200...0
```

- Hr menor que en otras políticas.

Asociativa

Un bloque de Mp puede ubicarse en cualquier bloque de Mca. Ej: $f = 0 \dots 511$ para todo i

Como en un bloque de Mca puede estar cualquier bloque de Mp hay i^{28} candidatos posibles, necesitamos 28 bits para identificar el bloque.

28 bits	4 bits
Etiqueta	Bytes

Hay que comparar la etiqueta de la dirección generada por la CPU con todas las etiquetas presentes en el directorio. Si alguna coincide acierto, si no fallo.

Ventajas

- Política de reemplazo muy flexible
- No hay fallos por conflicto: mayor Hr

Inconvenientes

- No se puede simultanear el acceso al directorio y al bloque, t_{acierto} mayor.
- Mayor coste: elevado nº de comparadores y mayor complejidad.

Asociativa por Conjuntos

Combinan las dos anteriores buscando la mejor de cada una.

Dividimos la Mca en conjuntos, grafos de bloques consecutivos y de tamaño fijo. Un bloque de Mp solo puede estar en un único conjunto (directa a nivel de conjuntos). Dentro de ese conjunto puede estar en cualquier bloque (asociativa dentro del conjunto).

Ej: supongamos que agrupamos nuestros 512 bloques de Mca de 2 en 2, tenemos 256 conjuntos de 2 bloques cada uno.

$$\frac{2^9}{2} = 2^8 \text{ conjuntos}$$

$$\frac{2^{28}}{2^8} = 2^{20} \text{ bloques se ubican en cada conjunto}$$

La etiqueta tendrá 20 bits. Para determinar el conjunto se usan 8 bits.

20 bits	8 bits	4 bits
Etiqueta	Conjunto	Byte

Accedemos al único conjunto donde puede estar el dato y comparamos nuestra etiqueta con todas las del conjunto. Si una coincide acierto, si no fallo.

Ventajas

- Reemplazo flexible a nivel de conjuntos
- Pocos comparadores. Coste bajo.
- Pocos fallos por conflicto

Desventajas

- Sigue teniendo fallos por conflictos.

Política de Extracción

¿Cuándo se lleva la información de Mp a Mca?

Hasta el momento hemos supuesto una política de extracción por demanda. La información se lleva a caché solo en caso de fallo. Esto implica que hay muchos fallos forzosos (1ª referencia).

La alternativa es una política con anticipación (prefetching). La idea es anticiparse a las necesidades del procesador llevando a Mca algunos bloques que previsiblemente se utilicen en un futuro.

La idea es explotar la proximidad de referencia espacial para anticiparse.

Obs: si fallamos en dicha predicción, llenamos la Mca de datos inútiles y puede hacer que disminuya el Hr.

Tipos posibles de extracción con anticipación

- **Siempre:** en cada referencia al bloque i trae el bloque $i+1$. Cada referencia se traduce en dos consultas en Mca (b_i, b_{i+1}). Costoso, no se utiliza.
- **Ante fallo:** si la referencia al bloque i genera un fallo nos traemos el bloque i y el bloque $i+1$. El bloque $i+2$ no se trae de forma anticipada y genera fallo.
- **Etiquetada:** si la referencia al bloque i genera un fallo nos traemos el bloque i y el bloque $i+1$, pero etiquetamos el bloque $i+1$. Si se hace referencia en un bloque etiquetado, bloque $i+1$, le quitamos la etiqueta y nos traemos el siguiente, bloque $i+2$, también con etiqueta.

La idea es reducir el nº de fallos forzosos y aumentar el Hr, pero puede ser incluso bajo si la predicción es mala. En cachés pequeñas, con poco espacio, no se usa. Además supone una penalización o un aumento del T_{fallo} ; nos traemos dos bloques.

También existe la política de extracción selectiva que consiste en dejar la opción de marcar información en Mp como no cacheable, de forma que no pueda viajar a Mca. Se puede usar en Multiprocesadores, E/S.

Tamaño Mca y de los bloques

Al aumentar la Mca disminuye la tasa de fallos ($1 - Hr$). Aumentar el tamaño de los bloques (con una Mca de tamaño fijo) puede aumentar o disminuir el Hr.

Bloques más grandes:

- *Favorecen proximidad de referencia espacial*
- *Perjudican* la posibilidad de *referencia temporal* (tenemos menos bloques). Cuando tengamos un nuevo bloque debemos reemplazar otro disminuyendo esta referencia y aumentando los fallos.
- *Aumenta el t_{fallo}* , sobre todo el tiempo de ocupación.

Solución: se busca un tamaño de bloque medio, según el comportamiento de los programas, tamaño Mca.. etc.

Política de Reemplazo

Determina el bloque que debe desalojar la Mca para hacer sitio a un nuevo bloque.

Sólo tiene sentido en Asociativa y Asociativa por Conjuntos.

Distintas alternativas:

- *Aleatoria*: sencilla y rápida
- *FIFO*: por tiempo de permanencia, estilo cola (First In First Out)
- *LRU*: se desaloje el que lleve más tiempo sin ser usado (Least Recently Used). Éste posee un buen Hr pero es complejo de implementar. En cada acceso a Mca (con acierto o fallo) debe actualizarse la información relativa a la gestión de la Mca.

Política de Escritura

Indica cómo realizar las escritura en Mca, existen dos opciones:

- **Escritura inmediata** (Write Through, WT)

Escribe simultáneamente en Mca y Mp. La información es coherente.

$$t_{\text{acceso}} = t_{\text{ocupación}} = \max(t_{Mca}, t_{Mp}) = t_{Mp}$$

Sencilla de implementar. Se suele combinar con buffers de escritura para reducir el tiempo de acceso.

- **Escritura aplazada** (Copy Back, CB)

Se escribe solo en Mca, la información entre Mp y Mca no siempre es coherente. Se necesita un "bit de modificado" por bloque.

Al reemplazar un bloque si éste ha sido modificado debe actualizarse la Mp para no perder información.

$$t_{\text{acceso}} = t_{\text{ocupación}} = t_{Mca}$$

Es más complejo de implementar.

En caso de fallo de escritura porque el dato no esté en memoria caché tenemos dos posibilidades:

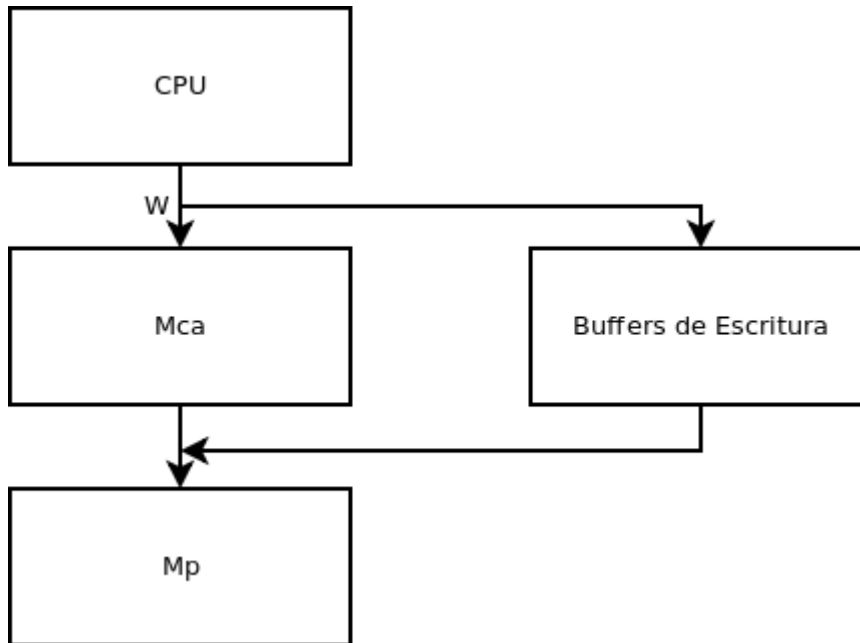
- **Con actualización de Mca** (with allocate)
- **Sin actualización de Mca** (with no allocate)

Normalmente:

- WT es con WNA → **WTWNA**
- CB es con WA → **CNWA**

Suponemos que el siguiente acceso al bloque sea similar: de escritura.

En CBWA normalmente se actualiza primero la Mp y luego se lleva el bloque a Mca. Comparando WT y CB, WT genera más tráfico a Mp (inaceptable en sistemas multiprocesador de mem.compartida), mantiene la coherencia, es más sencilla y tiene mayor t_{acceso} y $t_{\text{ocupación}}$.

Buffers de escritura

Se emplean normalmente con política WT para no tener que esperar a la Mp en cada acceso.

$$t_{acceso} = \max(t_{Mca}, t_{buffers}) = t_{Mca}$$

Se deja "anotada" la escritura en el buffer y la CPU puede continuar. El buffer, en paralelo realiza la escritura en Mp.

También se puede utilizar con CB, para reducir el tiempo en caso de fallo y que haya que reemplazar un bloque modificado.

Obs: si el buffer está lleno la CPU deberá esperar a que se haga hueco en el buffer.

Obs: cuando tenemos un fallo de lectura y el buffer no está vacío puede que el dato válido esté en el buffer y si accedemos a Mp podemos leer un dato incorrecto.

Para ello disponemos de posibles dos soluciones:

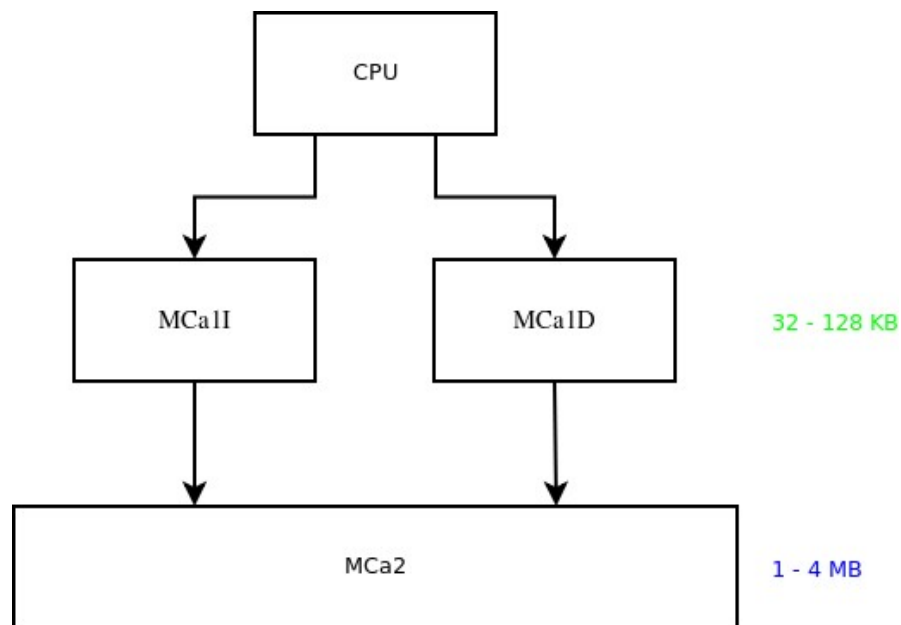
- _ Se espera a que se vacíe el buffer, lo cual es lento.
- _ Se consulta el contenido del buffer, esto aumenta la complejidad aunque aumenta la velocidad.

Cachés Unificadas vs Cachés Separadas

Podemos tener una caché unificada (instrucciones y datos juntos), como hemos supuesto hasta ahora. La alternativa es tener dos caches, una para instrucciones y otra para datos (arquitectura de Harvard). Los datos y las instrucciones tienen comportamientos distintos: las instrucciones son de sólo lectura, presentan mayor proximidad de referencia... etc.

Podemos particularizar las políticas de cada caché para adaptarnos mejor a su contenido y obtener un mayor Hr, a costa en un peor aprovechamiento del espacio (el espacio libre de una caché no la puede aprovechar la otra).

En cachés pequeñas funcionan mejor con cachés separadas (mayor Hr) mientras que en cachés grandes funciona mejor unificada. Las cachés de primer nivel (internas, en CPU) suelen estar separadas y las de segundo nivel (externas) unificadas.



Al estar separados se puede acceder simultáneamente a las dos Mca1, I y D algo imprescindible con procesadores segmentados (pipeline).

Cachés Multinivel

La idea es añadir un nivel más a la jerarquía de memoria. Se incorporan cachés internas, más pequeñas y rápidas, frente a cachés externas, más grandes y lentas.

Hay arquitecturas con 3 niveles de Mca (las 2 primeras, internas).

Ahora tenemos tasas de acierto y fallo locales a cada nivel además de globales.

$$(1 - Hr_{local \text{ nivel } i}) = \frac{\text{fallos nivel } i}{\text{accesos nivel } i}$$

$$(1 - Hr_{global \text{ nivel } i}) = \frac{\text{fallos nivel } i}{\text{accesos al S.M}}$$

Ej: la CPU emite 1000 peticiones y tenemos 40 fallos en Mca1 y 20 fallos en Mca2.

$$(1 - Hr_{local \text{ Ca1}}) = \frac{\text{fallos Ca1}}{\text{accesos Ca1}} = \frac{40}{1000} = 4 \%$$

$$(1 - Hr_{local \text{ Ca2}}) = \frac{\text{fallos Ca2}}{\text{accesos Ca2}} = \frac{20}{40} = 50 \%$$

$$(1 - Hr_{global \text{ Ca2}}) = \frac{\text{fallos Ca2}}{\text{accesos SmM}} = \frac{20}{1000} = 2 \%$$

En el segundo nivel es más significativa la tasa de fallos global que la local.

$$Tiempo \text{ efectivo} = Hr_{Ca1} \cdot T_{Ca1} + (1 - Hr_{Ca1})$$

$$Tiempo \text{ fallo Ca1} = [Hr_{Ca2}(T_{Ca1} + T_{Ca2}) + (1 - Hr_{Ca2})T_{Ca1} + T_{fallo \text{ Ca2}}]$$

Obs: Normalmente la Mca1 y la Mca2 tienen el mismo tamaño de bloques para facilitar la gestión.

Políticas de Lectura

La idea es reducir el tiempo de espera en caso de fallo en Mca. Hasta el momento hemos supuesto:

$$t_{Mca} \text{ Fallo} \quad t_{Mp} + n^{\circ} \text{ datos bloque} \quad t_{Mca1}$$

Para reducir el tiempo se emplea una estrategia impaciente llevando el dato a CPU en cuanto lo tengamos.

- **Early Start:** ¿3^{er} dato de un bloque? Con Fallo

CPU				3	
Mca		1	2	3	4
Mp	1	2	3	4	

En cuanto tenemos el dato lo llevamos a la CPU.

- **Out of order** (fetch COOF)

CPU		3			
Mca		3	4	1	2
Mp	3	4	1	2	

Cambiamos el orden de lectura.

La CPU continúa ejecutando, pero Mp y Mca siguen ocupadas terminando de transferir el bloque. Se emplean Mca no bloqueantes que permiten el acceso simultáneo desde Cpu y Mp (la CPU puede acceder mientras la Mca está actualizándose) excepto al bloque estamos analizando.

Ahora el t_{acceso} es bastante menor al $t_{\text{ocupación}}$.

t_{acceso} : tiempo que tarda la CPU en reanudar la ejecución.

$t_{\text{ocupación}}$: tiempo que emplea el SM en transferir un bloque.

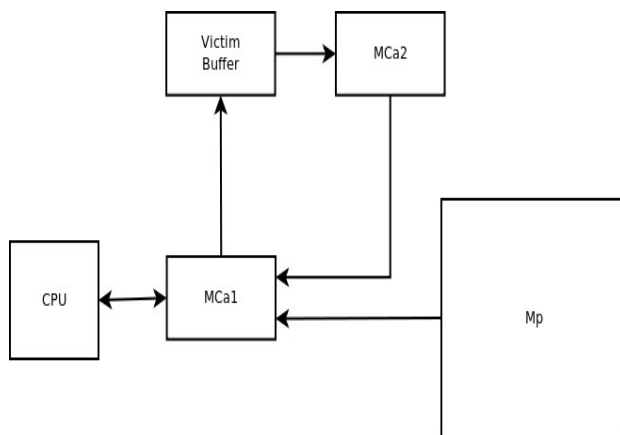
Si se produce un segundo fallo de Mca durante el tiempo de ocupación, la Mp no puede atender nuestra petición, debemos esperar a que termine con el bloque.

Obs: con los procesadores superescalares (ejecución de varias instrucciones simultáneamente, normalmente fuera de orden) esta restricción se suaviza.

Cachés de víctimas (victim buffer)

Es una pequeña caché (unos cuantos bloques 8-16) donde se guardan los bloques que se han desalojado de Mca. Es frecuente desalojar bloques que luego se necesitan.

Ej: Athlon



La caché de 2º nivel es una caché de víctimas, $L1 \not\subset L2$, $L1 \subset Mp$

Como $L1 \not\subset L2$, tenemos $128 + 256 = 384$ KB de Caché. Los bloques que se desalojan de Mca1 van al victim buffer, que poco a poco los vuelca a Mca2.

3. Memoria Principal

Memoria Principal entrelazada

$t_{\text{acceso}} \text{ o } t_{\text{latencia}}$ = tiempo que tarda en completarse una Op. De L/E.

t_{ciclo} : tiempo desde que la Mp comienza a atender una petición hasta que puede atender la siguiente.

Ancho de banda: cantidad de información transferida a/desde Mp por unidad de tiempo (bytes/seg).

La idea de la Mp entrelazada es aumentar el ancho de banda (AB) empleando memorias “normales” por tanto baratas. Se construye la memoria principal a base de una serie de módulos a los que se puede acceder de forma concurrente leyendo o escribiendo varias palabras a la vez, tantas como módulos tengamos.

El ancho de banda máximo o teórico viene dado por el AB convencional X nº de módulos. No siempre se consiguen, por ejemplo: dos accesos al mismo módulo deben hacerse de forma secuencial (colisión).

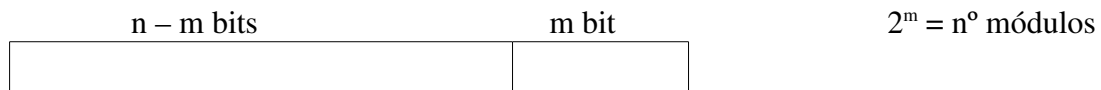
Este esquema es útil para:

- **Procesadores Superescalares** → Orden Inferior
- **Procesadores Vectoriales** → Orden Inferior
- **Memoria Caché** → Orden Inferior
- **Multiprocesadores** → Orden Superior

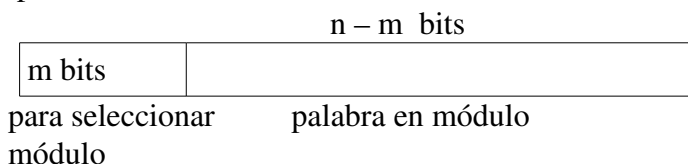
Es importante cómo se organiza el mapa de direcciones entre los distintos módulos, para evitar colisiones y maximizar el AB.

- **Entrelazado de orden inferior**: direcciones consecutivas en módulos distintos.
- **Entrelazado de orden superior**: direcciones consecutivas en el mismo módulo.

Dir orden inferior



Dir orden superior



Ej: 4 módulos utilizando Orden inferior.

Mod 0	Mod 1	Mod 2	Mod 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
.....

$$2^2 = 4 \text{ unidades, } m = 2$$

Mod 0	Mod 1	Mod 2	Mod 3
0	2^{n-2}	$2 \cdot 2^{n-2}$	$3 \cdot 2^{n-2}$
...
$2^{n-2} - 1$	$2 \cdot 2^{n-2} - 1$	$3 \cdot 2^{n-2} - 1$	$4 \cdot 2^{n-2} - 1$

Entrelazado simple

Se accede a todos los módulos utilizando la misma dirección. Implica entrelazado de orden inferior. Normalmente se emplea con un bus con capacidad para realizar transferencias de bloques.

Entrelazado complejo

Se puede acceder con direcciones distintas a cada módulo, suele tener un registro de dirección para almacenar su dirección. No implica ni entrelazado superior o inferior. El bus suele ser de ciclo partido, básicamente nos permite realizar una petición a memoria antes de haberse completado el anterior (el bus no está ocupado durante el acceso a Mp).

Ej: entrelazado inferior, 8 módulos

```
for(i=0;i<1000;i+=3)
    v=v+a[i]
```

A partir de la posición 0 de Mp.

$t_{Mp} = 100 \text{ ns}$

$$AB_{ideal} = \frac{8 \text{ módulos} \cdot 4 \text{ bytes/pal}}{100 \text{ ns}} = 320 \text{ MB/s}$$

Dir	0	3	6	9	12	15	18	21	24	27	30	33
mod	0	3	6	1	4	7	2	5	0	3	6	1

$$\text{Entrelazado simple (misma dir)} = \frac{3 \text{ mód} \cdot 4 \text{ bytes}}{100} \text{ ns} = 120 \text{ MB/s}$$

$$\text{Entrelazado complejo (distinta dir)} = \frac{8 \text{ mód} \cdot 4 \text{ bytes}}{100} \text{ ns} = 320 \text{ MB/s}$$

4. Memoria Virtual

Introducción

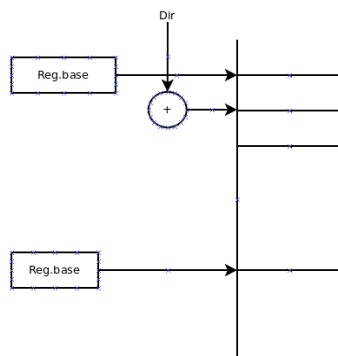
La Mp es un recurso único y escaso.

En caso de multiprogramación (lo normal) tendrá que compartirse entre varios procesos (¡es única!). Esto introduce la necesidad de protección y reubicación. Antes la protección se conseguía con los registros frontera (límite superior e inferior para todos los accesos a Mp de un programa).

`bz $etiq → desplazamiento en $C \pm 1$`

En todos los accesos debía comprobarse que no se sobrepasaban los límites, y esto era lento.

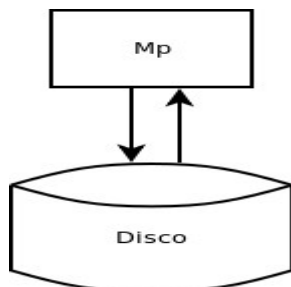
Antes la reubicación se conseguía con un registro base que marcaba la dirección de comienzo del programa. Había que sumar su contenido a todos los accesos a Mp.



Para programas “grandes” que necesitaban más memoria que la que había en la máquina se usaban “overlaps”. Se dividía el programa en fragmentos más pequeños y se incluían instrucciones de E/S para desalojar a disco unos fragmentos y traer de disco otros.

Es manual, lo hace el programador y cambia cada vez que cambia el HW.

Para solucionar estos problemas aparece la memoria virtual. El principal objetivo es proporcionar la “ilusión” de una capacidad ilimitada, pero también proporciona protección y reubicación.



Se emplea el disco duro como memoria, aumentando la capacidad de ésta. Vemos en el diagrama que la capacidad “aparente” es la del disco.

En Mp sólo tenemos un subconjunto de la información, la que está en uso, el resto está en el disco. Funciona por la propiedad de proximidad de referencia temporal y espacial.

Es decir, la memoria principal actúa como una memoria “caché” del disco.

La CPU accede a la información a través de la Mp (obviamos la Mca).

- _ Si acierto: fin.
- _ Si fallo accedemos a disco y nos llevamos la información a Mp.

El disco es **muy lento** y supone un problema: **cambio de contexto**, pasa a ejecutar otro proceso que esté listo. Además hay que definir políticas de reubicación, de reemplazo, de escritura... son propias del sistema operativo.

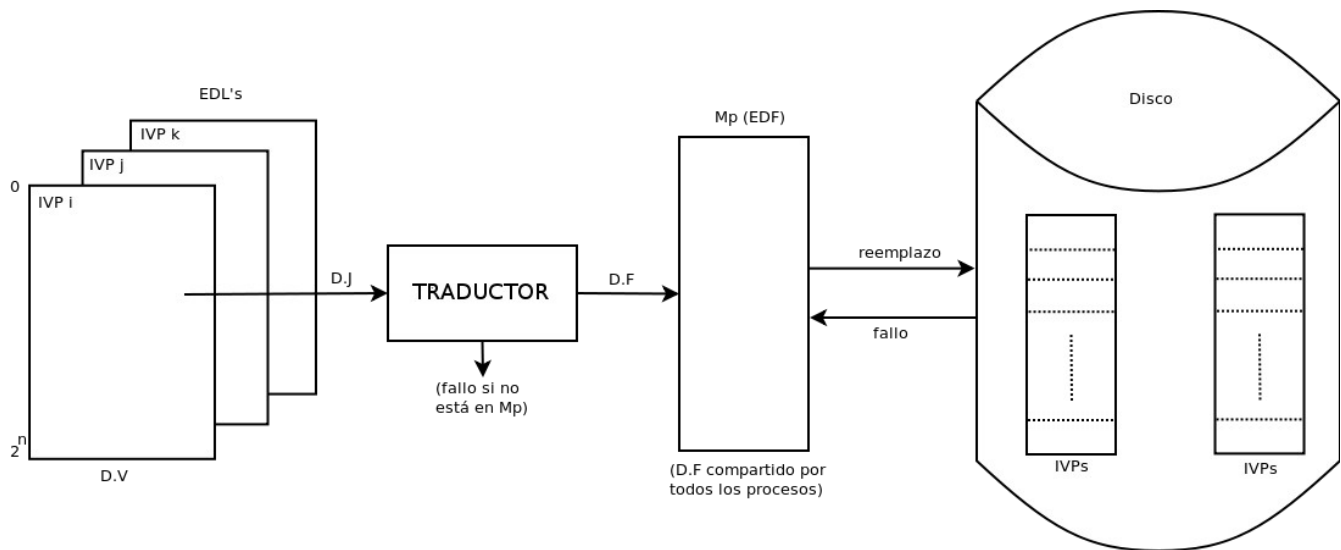
La idea básica de la memoria virtual (M.V) es distinguir entre el espacio de direcciones lógico (EDL) y el espacio de direcciones físicas (EDF).

- _ EDL: es el conjunto de direcciones que puede generar el juego de instrucciones de una arquitectura.
- _ EDF: la implementación, el conjunto de direcciones del que dispone la Mp de un computador.

Ej: i486, internamente maneja direcciones de 4 Gbits.

EDL: 2^{46} (64 TB)

El bus de direcciones tenía 32 bits, el EDF: 2^{39} (4 GB)



Cada proceso tiene su propio EDL independiente (no compartido) tiene la ilusión de que toda la memoria es suya y no la comparte. Este EDL constituye su imagen virtual (I.V) con direcciones virtuales (D.V).

Estas D.V deben traducirse en un “traductor” para obtener direcciones físicas (D.F) con las que acceden a Mp (o un fallo). En disco tenemos las IV de todos los procesos, llevando a Mp (compartida) sólo lo que está en uso. En cada acceso a memoria (Mp) debemos traducir la D.V a D.F, debe ser un proceso de traducción rápido.

Si la información no está en Mp, esta traducción genera un fallo en vez de la dirección física, este fallo hay que gestionarlo y traer la información de disco a memoria principal. El traductor avisa de que no ha podido hacer la traducción generando una excepción, el S.O responde ejecutando una rutina de tratamiento de excepción (RTE).

Esta rutina se encarga del cambio de contexto

- _ Salvar estado del proceso que generó el fallo (no puede seguir ejecutando)
- _ Aplicar política de reemplazo / ubicación
- _ Ordenar la transferencia de Disco a Mp (DMA)
- _ Restaurar el estado del nuevo proceso
- _ Pasar a ejecutar el nuevo proceso

Cuando se acaba la operación de E/S (DMA)

- _ Actualizar la información del traductor (ahora no debe generar fallo, sino la D.F donde se va a ubicar)
- _ Poner el proceso antiguo como ejecutable

Obs: Cuando el proceso que generó el fallo reanuda su ejecución tenemos dos posibilidades:

- _ Reiniciar la instrucción que generó el fallo
- _ Continuar con la instrucción que generó el fallo
- _ Reinicio de las instrucciones:

No se puede modificar el estado de la máquina hasta estar seguros de que no va a generar un fallo.

- _ Se puede trabajar sobre registros temporales y luego volcarlos al final de la instrucción.
- _ Usar direccionamientos “sencillos” (máquinas RISC)

Ej: Instrucción problemática: $L0 \cdot R1, \#18[++R2]$

- _ Continuar desde el mismo punto en que se generó el fallo. Implica salvar el estado (pila, μ PC, ...). Complicado y sólo se hace en algunas máquinas CISC.

Implementaciones de la memoria virtual

- _ *Paginación*
- _ *Segmentación*
- _ *Segmentación paginada*

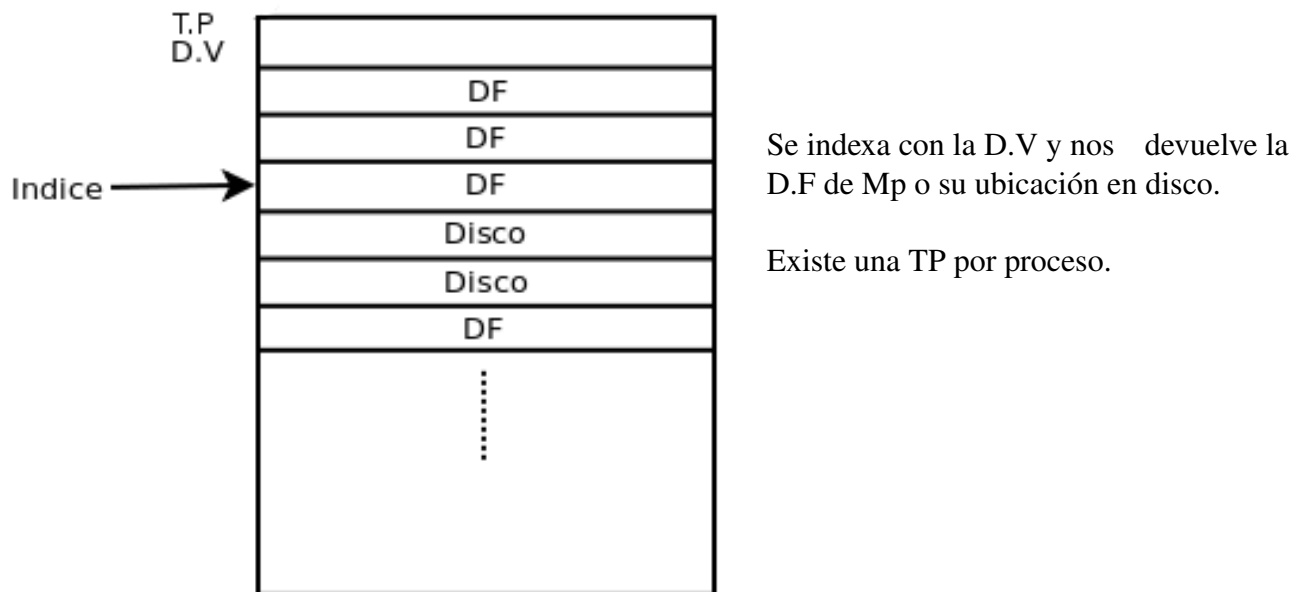
Paginación

La idea básica es dividir las EDL's y EDF en bloques de tamaño fijo (unidad mínima de información para la M.V)

Obs: es el equivalente a los bloques de Mca.

Estos bloques se llaman páginas en el EDL y marcos de página en el EDF. Son bloques “grandes” (comparado con los bloques de Mca) 8KB – 32KB para aprovechar mejor los accesos a disco aunque supone desperdiciar algo de Mp.

La traducción se hace en base a una Tabla de páginas:



Además existe un registro base de la Tabla de páginas (RBTP) único (para todos los procesos) que marca el comienzo de la T.P.

En cada cambio de contexto el S.O debe actualizarlo.

Obs: los bits de dirección relativos al desplazamiento dentro de la página no debe traducirse, son los mismos en la D.F que en la D.V.

Ubicación y tamaño de la Tabla de Páginas (TP)

La ubicación será en Mp. El tamaño depende de cada caso.

Ej: DV 46 bits

páginas de 16 KB

palabras de 4KB

4B/entrada TP

$$\frac{2^{46} \text{ Bytes}}{2^{14} \text{ Bytes/pág}} = 2^{32} \text{ páginas}$$

$$2^{32} \text{ pag} \cdot 4 \text{ Bytes/entrada} = 2^{34} \text{ bytes} = 16 \text{ GB (ocupa la TP)}$$

Serían 16 GB por TP y hay una por proceso. Imposible. Se divide la TP en fragmentos más pequeños.

- _ TP paginada: no se usa. Es del VAX (DEC). Difícil de gestionar.
- _ TP multinivel: dividimos las páginas en zonas y tenemos una TP del P nivel para acceder a las zonas y luego una TP del 2º nivel para traducir las D.V de las páginas de cada zona. El RBTP apunta a la TP de 1º nivel, que es la única que necesitamos en Mp, las demás se pueden traer bajo demanda.

Actualmente ya existen muchas arquitecturas con incluso tres niveles de tablas de páginas.

Ej: 4 Gbits de D.V

zona		46 bits
TPN1	TPN2	páginas
16 bits	16 bits	14 bits

La TPN1 ocupa 2^{16} entradas \cdot 4 bytes/entrada = 2^{18} = 256 KB

Las demás, TPN2, ocupan lo mismo y se traen por demanda.

El problema de este esquema es el tiempo de traducción que se necesita (2 ó 3 accesos a Mp, según el nº de niveles).

Soluciones: emplear una “caché” para reducir este tiempo: TLB (Translation Lookaside Buffer, una caché de las tablas de traducción).

Contiene la información de las últimas traducciones realizadas. La TLB suele tener pocas entradas porque se manejan pocas páginas en un programa (32 – 128 entradas). Se busca en la TLB por la D.V, si se encuentra es acierto y se recupera la información de traducción. Si se produce fallo (no está en TLB) se accede a las TP's de Mp (lento) y se actualiza la TLB para los siguientes accesos.

Los fallos de TLB (pocos y lentos de gestionar) se tratan por:

- _ HW: en la Mca
- _ SW: el S.O trata los fallos de TLB. Es más flexible.

Las TLB's normalmente son asociativas o asociativas por conjuntos (pol.ubicación). La política de reemplazo suele ser LRV y la política de escritura CBWA (algunas TLB no permiten modificar los datos).

Obs: debido a la naturaleza distinta de datos e instrucciones, actualmente se distingue entre TLB de instrucciones y TLB de datos. Además algunos sistemas tienen TLB's de 2º nivel (TLB's multinivel).

- 32 – 128 entradas: 1º nivel
- 512 – 1024 entradas: 2º nivel

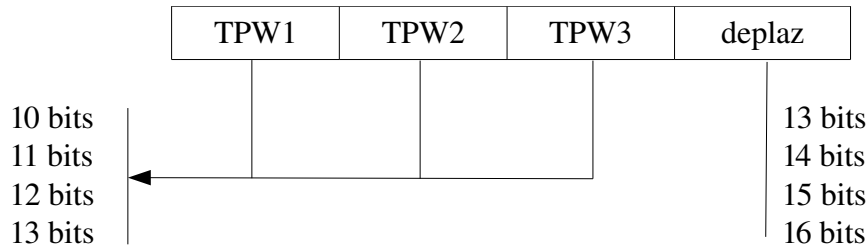
Obs: Problemas de la TLB en los cambios de contexto.

Como los procesos tienen las mismas D.V se pueden producir aciertos casuales (porque el proceso previo ve la misma D.V). Para evitar esto hay que invalidar la TLB para que inicialmente esté vacía.

Pero invalidar la TLB es lento y más lento aún es recargar la TLB (muchos fallos de TLB) al principio.

La solución es añadir el PID de cada proceso a las entradas de la TLB para evitar aciertos casuales.

Ej: Alpha D.V de hasta 64 bits 3 niveles de TP's



Cada entrada de la TP ocupa 8 bytes, cada TP ocupa una página.

43 bits	TLB _d : 64 entradas	Fallos gestionados por SW
47 bits	TLB _i : 48 entradas	No permite modificar los datos de la TLC
51 bits		
55 bits		

Obs: PowerPC

Otro posible esquema de traducción es emplear traducción inversa mediante una tabla hash. Sólo hay entradas para las Dir realmente usadas(mucho más pequeño que las TP's). Con la D.V se entra en la tabla hash y se recupera la D.F.

Segmentación

En la paginación no se tiene en cuenta la naturaleza de la información. La alternativa es agrupar la información del mismo tipo (pila, código, datos globales, datos dinámicos, ...) en “segmentos”. En vez de páginas pasamos a tener segmentos con información homogénea,

Estos segmentos pueden ser muy grandes y de tamaño variable y son difíciles de gestionar.

34 bits	30 bits
segmento	Deplazamiento

Hay pocos segmentos (1^{er} campo pequeño) y son grandes (2º campo grande) por lo que son poco manejables. Por ejemplo en caso de fallo de segmento habría que traerse TODO el segmento, no es factible.

Además un segmento puede crecer (pila) y necesitar reubicación, lo que implica mover un gran volumen de información en la memoria.

La ventaja que tienen es que al tener pocos segmentos la tabla de segmentos es pequeña, tan pequeña que hasta se puede ubicar la información de traducción en los registros del procesador. (traducción muy rápida).

Al ser información homogénea los mecanismos de protección y compartición son más naturales y sencillos.

Segmentación paginada

Para solucionar los problemas de gestión, los segmentos (reubicación como llevar / traer a disco todo el segmento) dividimos los segmentos en páginas de tamaño fijo y sólo tendremos en Mp las páginas en uso.

Ahora hay 2 niveles de traducción: 1^{er} nivel : Tabla de segmentos
2º nivel: Tabla de páginas.

DV

Seg	Pág.Virtual	deplaz
-----	-------------	--------

Se emplea TLB y las TP's son multinivel.

Combinación de Mca y M.V

Primero se traduce la D.V a D.F y luego se emplea la D.F para acceder a Mca, si hay fallo nos vamos a Mp. Al concatenar varios pasos se alarga el tiempo.

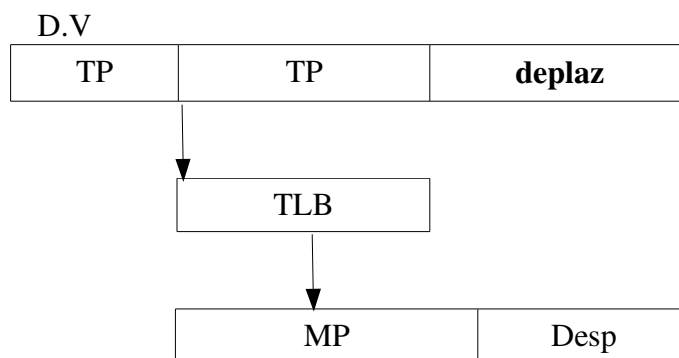
Dos soluciones:

- _ Solapar el acceso a la TLB y a la Mca
- _ Emplear Mca virtuales

Solapar acceso a TLB y Mca

Para paginación

En negrita: no se traduce ya es D.F. Se puede emplear para acceder a la Mca.



Directa

Etiqu.	Bloq.	deplaz
--------	--------------	---------------

Asoc.Conjunto

Etiqu.	Conj.	deplaz
--------	--------------	---------------

El tamaño de la página (parte que no se traduce de la D.V) debe ser igual o mayor al nº de bits que necesito para acceder al bloque + byte (en directo) o al conjunto +byte (en Asoc.Conj)

Sólo será posible en Mca pequeñas (las de 1^{er} nivel)

Mca Virtuales

Son cachés que trabajan directamente con D.V en vez de D.F. Nos ahorramos el paso de traducción, necesario ahora sólo en caso de fallo en Mca (**la Mp no conoce las D.V**)

Inconvenientes:

- _ Las D.V son las mismas para todos los procesos por lo que pueden darse aciertos casuales.
- _ La solución es invalidar la Mca (costoso, no) o añadir el PID a la Mca. La misma D.V corresponde a distintas D.F según el proceso.
- _ Sinónimos: Información compartida entre varios procesos de modo que D.V distintas corresponden a la misma D.F

Soluciones:

- Que el S.O evite los sinónimos
- Que la información compartida no sea cacheable. Sencilla, pero no es admisible en multiprocesadores
- Alinear los últimos bits de la D.V de los sinónimos en el mismo bloque de Mca (Válida en Mca directas)
- Alpha 21264: emplea Mca virtuales, emplea etiquetas físicas (no virtuales), accede a la Mca con la D.V, pero la comparación para ver si ha tenido éxito o fallo la hace con la etiqueta física. Nos obliga a traducir todas las D.V, pero permite solapar el acceso a la Mca (D.V) y la traducción en la TLB.